Bitmasking

Bit-masking is a technique to selectively modify individual bits without affecting other bits.

Bit SET

To set a bit, we need to use the OR operator. This is just like an OR logical gate you should've learnt in Digital Design course. To set a bit, you would OR a memory with a bit number and the bit number with which you will OR will end up getting set.

```
// Assume we want to set Bit#7 of a register called: REG
REG = REG | 0x80;

// Let's set bit#31:
REG = REG | 0x80000000;

// Let's show you the easier way:
// (1 << 31) means 1 gets shifted left 31 times to produce 0x80000000
REG = REG | (1 << 31);

// Simplify further:
REG |= (1 << 31);

// Set Bit#21 and Bit# 23:
REG |= (1 << 21) | (1 << 23);
```

Bit CLEAR

To reset or clear a bit, the logic is similar, but instead of **OR**ing a bit, we will AND a bit. **Remember that AND gate clears a bit if you AND it with 0 so we need to use a tilde (~) to come up with the correct logic:**

```
// Assume we want to reset Bit#7 of a register called: REG
REG = REG &   0x7F;
REG = REG & ~(0x80); // Same thing as above, but using ~ is easier

// Let's reset bit#31:
REG = REG & ~(0x80000000);

// Let's show you the easier way:
REG = REG & ~(1 << 31);

// Simplify further:
REG &= ~(1 << 31);

// Reset Bit#21 and Bit# 23:
REG &= ~( (1 << 21) | (1 << 23) );
```

Bit TOGGLE

```
// Using XOR operator to toggle 5th bit
REG ^= (1 << 5);

// Invert bit3, and bit 5
REG ^= ((1 << 3) | (1 << 5));
```

Bit CHECK

Suppose you want to check bit 7 of a register is set:

```
if(REG & (1 << 7))
{
    DoAThing();
}

// Loop while bit#7 is a 0
while( ! (REG & (1 << 7)) ) {
  ;
}
```

Now let's work through another example in which we want to wait until bit#9 is 0

```
// As long as bit#9 is non zero (as long as bit9 is set)
while((REG & (1 << 9)) != 0) {
  ;
}

// As long as bit#9 is set
while(REG & (1 << 9)) {
  ;
}
```

GPIO Example of NXP CPU

In this example, we will work with an imaginary circuit of a switch and an LED. For a given port, the following registers will apply:

- GPIO selection: PINSEL register (not covered by this example)
- GPIO direction: DIR (direction) register
- GPIO read: IOPIN register
- GPIO write: IOPIN register

Each bit of FIODIR1 corresponds to each external pin of PORT1. So, bit0 of FIODIR1 controls direction of physical pin P1.0 and bit31 of FIODIR2 controls physical pin P2.31. Similarly, each bit of IOPIN1 or IOPIN2 controls output high/low of physical ports P1 and P2. IOPIN not only allows you to set an output pin, but it allows you to read input values as sensed on the physical pins.

Suppose a switch is connected to GPIO Port P1.14 and an LED is connected to Port P1.15. Note that if a bit is set of FIODIR register, the pin is OUTPUT otherwise the pin is INPUT. **So... 1=OUTPUT, 0=INPUT**

```
// Set P1.14 as INPUT for the switch:
LPC_GPIO1->DIR &= ~(1 << 14);

// Set P1.15 as OUTPUT for the LED:
LPC_GPIO1->DIR |=  (1 << 15);

// Read value of the switch:
if(LPC_GPIO1->PIN & (1 << 14)) {
    // Light up the LED:
    LPC_GPIO1->PIN |= (1 << 15);
} else {
    // Else turn off the LED:
    LPC_GPIO1->PIN &= ~(1 << 15);
}
```

LPC also has dedicated registers to set or reset an IOPIN with hardware AND and OR logic:

```
if (LPC_GPIO1->PIN & (1 << 14)) {
    LPC_GPIO1->SET = (1 << 15); // No need for |=
}
else {
    // Else turn off the LED:
    LPC_GPIO1->CLR = (1 << 15); // No need for &=
}
```

Brainstorming

- How many ways to test an integer named **value** is a power of two by using a bit manipulation?

```
(value | (value + 1)) == value
(value & (value + 1)) == value
(value & (value - 1)) == 0
(value | (value + 1)) == 0
(value >> 1) == (value/2)
((value >> 1) << 1) == value
```

- What does this function do?

```
boolean foo(int x, int y) {
  return ((x & (1 << y)) != 0);
}
```